del Rio Astorga D, Dolz MF, Sánchez LM, Fernández J, García JD. An adaptive offline implementation selector for heterogeneous parallel platforms. *The International Journal of High Performance Computing Applications*. 2018;32(6):854-863

DOI: 10.1177/1094342017698746

An adaptive offline implementation selector for heterogeneous parallel platforms

David del Rio Astorga[1], Manuel F. Dolz[1], Luis Miguel Sanchez[1], Javier Fernández[1], and J. Daniel García[1]

[1] *Universidad Carlos III de Madrid, 28911–Leganés, Spain*

Heterogeneous parallel platforms, comprising multiple processing units and architectures, have become a cornerstone in improving the overall performance and energy efficiency of scientific and engineering applications. Nevertheless, taking full advantage of their resources comes along with a variety of difficulties: developers require technical expertise in using different parallel programming frameworks and previous knowledge about the algorithms used underneath by the application. To alleviate this burden, we present an adaptive offline implementation selector that allows users to better exploit resources provided by heterogeneous platforms. Specifically, this framework selects, at compile time, the tuple device-implementation that delivers the best performance on a given platform. The user interface of the framework leverages two C++ language features: attributes and concepts. To evaluate the benefits of this framework, we analyze the global performance and convergence of the selector using two different use cases. The experimental results demonstrate that the proposed framework allows users enhancing performance while minimizing efforts to tune applications targeted to heterogeneous platforms. Furthermore, we also demonstrate that our framework delivers comparable performance figures with respect to other approaches.
[1]

1 Introduction

In recent years, heterogeneous parallel architectures have provided a way to improve performance and energy efficiency better than other alternatives. However, platforms comprising diverse devices (such as multi-cores, GPUs, DSPs and FPGAs) are notoriously more difficult to program effectively, since they demand for distinct frameworks and application programming interfaces [5]. This fact, has led to multiple implementations of the same algorithm but targeted to different devices. Therefore, an additional issue arises when programming for heterogeneous platforms: to select the most suitable device and routine implementation to solve a given problem. Usually, in order to improve performance, developers need to analyze a priori the target platform and the application, along with its implementation alternatives and available libraries. To achieve this goal, some aspects need to be considered. For instance, some libraries exhibit better behavior than others for a given problem size [10]. Also,

[1]This is a post-print of an article published in The International Journal of High Performance Computing Applications. The final authenticated version is available online at: https://doi.org/10.1177/1094342017698746

devices can have different features (such as the number of cores, processor frequency or memory size), and thus, they may influence, or even restrict, the use of a specific library routine.

An approach to cope with this problem is to manually select the algorithm implementation and map the execution onto the underlying parallel device based on past knowledge. Nevertheless, this procedure becomes complex when dealing with multiple devices and libraries. A common technique is to define a set of constraints in order to guide a runtime scheduler to select the most suitable implementation. This technique, however, has non-negligible performance overheads, since it is necessary to reevaluate the selection each time a routine is called. An alternative to the aforementioned technique is to shift the decision-making task directly at compile time. Several proposals leveraging this static approach and based on analytic models, machine learning and adaptive optimization methods can be found in the literature [1]. However, these approaches also incur in non-negligible profiling costs at run-time, and/or modeling overheads at compile time. Therefore, there exists a trade-off between runtime overheads and profiling techniques in both approaches that needs to be considered depending on the target application. Given the foregoing, in this paper we enrich the current state-of-the-art about static approaches with the following contributions:

- We present an offline implementation selector framework that leverages a profile-guided approach and is able to decide, among compilations, the tuple device-implementation that delivers the best performance based on historical information.
- We enable portability of the approach by using two novel features part of the standard C++ language, concepts and attributes, as for the end-user interfaces.
- We evaluate the performance of the selector by analyzing its convergence time and benefits to minimize execution time using the general matrix-matrix multiplication kernel and an image-processing application.
- We demonstrate that the framework self-tunes to platform changes and delivers comparable performance figures with respect to a runtime approach from the state-of-the-art.

The remainder of the paper is structured as follows. Section 2 reviews existing works about implementation selectors and decision-making models. Section 3 describes the concepts and attributes C++ language features along with the hardware parallel platform description language. Section 4 introduces our adaptive offline implementation selector framework and its decision algorithm. Section 5 evaluates the convergence time and performance benefits of the adaptive selector and compares it with a runtime approach. Finally, Section 6 closes the paper with some concluding remarks and future works.

2 Related Work

Since heterogeneous platforms have spread across the scientific community, different implementations of the same algorithm have been developed for specific devices. For example, several numerical libraries comprising highly tuned kernels, from BLAS and LAPACK, are available for several computing architectures, e.g., cuBLAS [7] for nVidia GPUs, GSL [4] for multi-/many-core processors, etc. This situation reveals a new challenge: to select the most suitable device and routine implementation to solve a given problem. To tackle this issue, two different approaches have been traditionally taken: *i)* runtime schedulers, able to map and execute kernels from multiple libraries on accelerators available in a heterogeneous platform, and *ii)* static tools that select, at compile time, the most appropriate implementation according to past knowledge. Obviously, dynamic approaches are more flexible in terms of mapping tasks to devices according, e.g., on load parameters, while static approaches have to use concrete implementations during the entire application run.

Some research works using static approaches can be found in the literature. For instance, the work by Zheng Wang et al. [15] presents a mapping model based on machine learning techniques, that is tied to the training platform. Other works, as presented by Jun et al. [14], propose an automatic system based on source code analysis that maps user calls to optimized kernels. Similarly, Jie Shen et al. [11] propose an analytic system for determining which hybrid programming configuration is optimal for a given problem.

On the other hand, dynamic approaches are also greatly extended in the community. Particularly, the OmpSs [3] programming framework leverages an extended set of OpenMP-like pragmas to support asynchronous parallelism and exploit task-parallelism of applications via data-dependencies. Concretely, among the pragma options, the `target` directive allows developers to select the target device in an heterogeneous platform. Together with this directive, the `implements` clause lets users to specify that the annotated code is an alternate implementation of a given function for a specific device. This feature allows its *versioning* runtime scheduler to freely map the same task onto different devices. Other works in this line, such as the extension for the SkePu framework presented in [2], take advantage of a machine learning mechanisms in order to select automatically the most appropriate implementation of a given function. These models basically carry out a tuning phase to estimate the ranges in which different implementations perform better than others and to take the most appropriate one. Following a similar approach, the tool presented in this paper, leverages C++ attributes and concepts to select between

implementations and devices from the heterogeneous platform. However, it shifts the selection process directly at compile time. Obviously, this approach restricts the flexibility but offers benefits in terms of memory usage for embedded systems.

3 Background

This section gives a brief overview about the two C++ language features used for developing the implementation selector interface: C++ attributes and concepts. Furthermore, we describe the hardware parallel platform description language leveraged by the selector to keep track of available resources.

3.1 The C++11 attributes

Attributes are a new feature of the C++11 language for defining properties to programming entity units. The power of the attributes is that they provide extra information to the compiler in order to perform a given action to the entity that has been annotated. One of the main advantages of the attributes with respect to pragmas, is their flexibility, as they can provide properties to any entity, e.g. variables, functions or types, and do not need to appear on a separate line [6]. The basic syntax for attributes in C++11 is defined with: `[[namespace::attribute]]`.

However, unlike attributes from other languages, C++11 attributes are compiler-defined, i.e., the user cannot define their own attributes at runtime. Indeed, the C++ Run-Time Type Identification (RTTI) mechanism does not keep any attribute information, so the knowledge given in the attributes is not accessible from the user application. On the contrary, attribute extensions require modifications in the C++ compiler. Therefore, the purpose of this feature is to allow future C++ extensions without extending the set of keywords nor grammar.

3.2 The C++ concepts

Concepts are a novel extension of the C++ programming language that allows defining and evaluating, at compile time, constraints set on template arguments [12]. Particularly, concepts deliver a better support for error checking in generic programming contexts, thus, they can be seen as a mechanism to prevent and diagnose improper uses of templates. In general, concepts allows to overloading templates and disabling those whose types do not satisfy the predefined constraints. Lexically, concepts can be expressed with the keywords `concept` and `requires`. Their current implementation can be found in the GCC C++ experimental branch, namely *concepts lite* [13].

To enable portability of our framework, we use both attributes and concepts as for the user interfaces. However, given that the concepts are a novel feature of the language, they are not yet supported by all the current C++ compilers. In contrast, attributes are a feature of the C++ language that is already supported by any compiler. In a nutshell, attributes are easier to use than concepts, but at the expense of modifying the compiler internals or using an external parsing tool to process them.

3.3 The hardware parallel platform description language

The Hardware Parallel Platform Description Language (HPP-DL) is an open specification that leverages hierarchical models for describing features of Heterogeneous Parallel Platforms (HPP) [8]. This language is intended to be used for making platform-specific information available to developers and tools, such as auto-tuners, compilers, schedulers or runtime systems.

This specification comprises information about the following classes: *i) components* describe the hardware resources of the platform, involving processors (CPU sockets), cores, main memory, GPUs or OpenCL based devices; *ii) links* represent relationships between two *components* in one way and incorporate information about data transmission: throughput and latency; and *iii) resources* portray interfaces for accessing to computing devices attached, e.g. FPGAs or DSPs. These resources comprise I/O ports, IRQs or address ranges.

4 The adaptive offline implementation selector

In this section we describe the adaptive and offline implementation selector framework as the main contribution of this paper. Figure 1 shows the general workflow of this framework. As can be seen, in a first step the system administrator provides users with all the possible implementations of a function by producing, using the header generation tool, all necessary function wrappers for the two proposed interfaces, i.e., attributes and concepts. This command-line tool takes, for each function to be replaced, the following arguments: *i*) the function name; *ii*) all the available implementation names (e.g., GSL or clBLAS); and *iii*) the supported devices (e.g., CPU or

GPU). With this information, the command-line tool generates the wrappers whose name is a concatenation of the three aforementioned arguments. Afterwards, the function wrappers in the generated header files should be completed by the administrator with the corresponding code for calling to specific implementations. Note that all generated headers are automatically instrumented for measuring execution time. This is done in order to support the profile-guided approach implemented by the selector. Additionally the headers generated incorporate the required code to support the framework interfaces. In parallel, the administrator should also obtain required hardware information, using the automatic tool provided for storing it into a `HPP.json` file, according to the HPP-DL specification.

On the other hand, users should call the interfaces, generated by the administrator and present in the header files, and provide static information (e.g. problem size) to the selector in the form of attributes or template arguments. Next, the attribute and concepts-based implementation selectors, call respectively to the selector module so as to determine the most suitable implementation based on historical data and the static information provided by the user. Finally, once the application has been compiled and executed, the performance file (`PERF.h`) is updated automatically with the new performance data (e.g. "problem size–execution time" tuples) in order to improve the selector knowledge in future decisions.

In the following sections, we describe how the concepts and the attribute mechanisms should be used within the framework. Next, we enumerate the steps taken by the selector module.

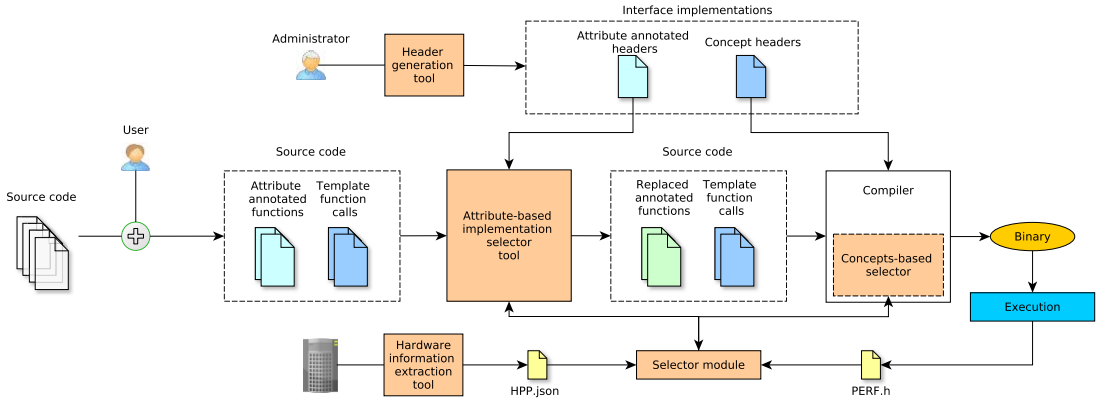


Figure 1: Diagram of the adaptive offline implementation selector.

4.1 The attributes-based interface

The interface of the framework based on C++ attributes is intended to define constraints in order to guide the compiler to select a concrete implementation among the available ones. These attributes are part of our previous work presented in [9].

Administrator actions. First, the administrator needs to run the header generation tool, with the *attributes* flag option set, so as to generate the headers containing the annotated function wrappers for a given algorithm. Listing 1 shows an example of generated code for the `dgemm` kernel. Next, the administrator should complete the attribute-annotated wrapper bodies with the corresponding function calls for the interfaces generated. The attributes inserted, under the `ais` (adaptive implementation selector) *namespace*, are the following:

- `ais::implements`: This attribute specifies that the code under the attribute is an alternate implementation of a given interface. Basically, it receives, as a single parameter, the function name to let the selector know which implementations are available for that interface.
- `ais::device`: This attribute bounds a given implementation to a specific target device. Examples of valid parameters for this attribute are: `CPU`, `GPU`, `PHI` (for the Intel Xeon Phi), etc.

Listing 1: Generated header file for attribute-annotated function implementations.

```
1 [[ais::implements(dgemm), ais::device(CPU)]]
2 dgemm_clBLAS_CPU( ... ){  /* Call to clBLAS dgemm to be filled by the administrator */  }
3 [[ais::implements(dgemm), ais::device(CPU)]]
4 dgemm_GSL( ... ){         /* Call to GSL dgemm to be filled by the administrator */  }
```

Users actions. On the other hand, the users are responsible for annotating candidate function calls in the application code in order to be analyzed by the attribute-based implementation selector tool of the framework. These C++ attributes are the following:

- `ais::interface`: This attribute indicates that the annotated function call is an interface and should be treated by the tool and replaced during the selection process.
- `ais::target`: It defines the preferred target device to execute an annotated interface, e.g., `ais::target(CPU)`. Valid arguments for this attribute are those accepted by the `ais::device` attribute.
- `ais::size`: This attribute is used when the user knows a priori the problem size during the function call. As a single parameter, it receives an integer indicating the problem size.
- `ais::min` and `ais::max`: Alternatively, when the user does not know the specific problem size, these attributes can be used to let the selector know the lower and upper bounds of the same used to execute a certain function.

For instance, Listing 2 contains an example of user code with different C++ attribute-annotated function calls that match the interface `dgemm` defined by the administrator in the header file. As can be observed, both first and second calls to `dgemm` have been annotated using the attributes for a fixed size and a range of sizes, respectively. After the framework preprocesses the annotated source code, the interfaces are automatically replaced by the most suitable function implementations. In the example of Listing 3, both `dgemm` calls have been replaced by function calls to the clBLAS and GSL kernel implementations, respectively.

Listing 2: User-annotated function interfaces.

```
[[ais::interface, ais::size(1024)]]
dgemm( ... );
[[ais::interface, ais::min(256), ais::max(512)]]
dgemm( ... );
```

Listing 3: Replaced attributes by implementations.

```
// Replaced annotation by clBLAS-CPU dgemm
dgemm_clBLAS_CPU( ... );
// Replaced annotation by GSL dgemm
dgemm_GSL( ... );
```

4.2 The concepts-based interface

The interface of the framework based on concepts is an alternative to the aforementioned mechanism based on attributes that pursues the same goal.

Administrator actions. In this case, the system administrator uses the header generation tool with the *concepts* flag option set to introduce an algorithm implementation and the devices supported. Listing 4 shows an example of generated code for the `dgemm` kernel. This tool generates a header file containing the function templates with the concepts in charge of determining the most suitable implementation for a given input size or range of sizes. Next, the function templates in the header file are to be completed manually with the corresponding function calls to specific implementations for such an algorithm.

Listing 4: Generated header file for concepts function implementations.

```
template <int size> requires clBLAS_CPU(size)
dgemm( ... ){  /* Call to clBLAS dgemm to be filled by the administrator */  }
template <int size> requires GSL(size)
dgemm( ... ){  /* Call to GSL dgemm to be filled by the administrator */  }
}
```

User actions. Moreover, the users call the supported function interfaces providing the necessary information, in the form of template arguments, so as to guide the compiler through the framework to link against concrete implementations. As an example, the first call to function `dgemm` in Listing 5 provides a fixed problem size of the matrices, while the second provides a range of sizes. All in all, this static information allows the compiler to select the `dgemm` implementation thanks to the concepts introduced by the header generation tool and the requirements specified by the user.

Listing 5: Concepts mechanism used in the `dgemm` function interface.

```
dgemm</*size*/ 1024>( ... );            // This call will be linked against clBLAS-CPU dgemm
dgemm</*min*/ 256, /*max*/ 512>( ... ); // This call will be linked against GSL dgemm
```

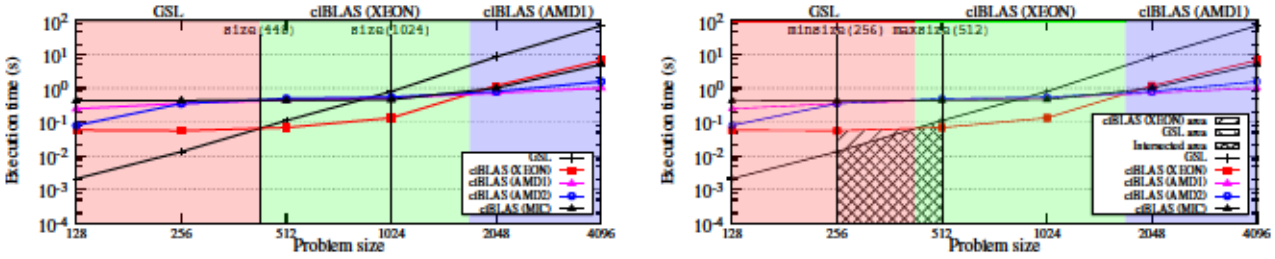
4.3 The selector module

This section describes the internal algorithm of the adaptive implementation selector module. This algorithm has been implemented for accepting the two aforementioned interfaces: attributes and concepts. Note that the attributes-based implementation selector has been developed as a preprocessing tool using the Clang 3.8.0 compiler API, while the concepts-based selector is implicitly embedded into the semantic concepts code and interpreted by the compiler itself.

The selection algorithm implemented by our framework is entirely based on the problem size and boundaries specified by the user. Depending on this information, the algorithm proceeds as follows:

- If the template argument or attribute for a fixed size is set, the selector takes the implementation offering the minimum execution time. This is done using the information stored in the `PERF.h`. To do so, the selector performs a linear interpolation for the requested problem size for all implementations available in such a function interface. Otherwise, if multiple implementations deliver the same minimum performance, the selector randomly picks one of them. Consider the scenario in Figure 2a showing the behavior of the `dgemm` interface offering five implementations. For instance, if the user sets the size parameter to 1,024, the selector will consider the clBLAS version running on CPU, while if the problem size is fixed to 448, the selector will randomly select GSL or clBLAS (Xeon). To avoid extreme behaviors in some cases, this random policy can be eventually replaced by another that takes into account the lower maximum performance.
- On the contrary, if the developer has indicated a range of possible problem sizes, the selector module computes the area under the performance curve (or integral) between the ranges for each implementation available. This information allows the selector to take the implementation that has the smallest area between the ranges. As shown in Figure 2b, if the user selects the range 256–512, the selector module will compute the integrals for the five implementations available. Afterwards, it will compare the areas below the curves and take that having the smallest area, i.e., GSL. Note that if there are no performance values in the range boundaries, the values that intersect the boundaries are computed via linear interpolation. As in the previous option using a fixed size, if there are two or more implementations whose integral value is equal, the selector will pick one randomly.

Note that, at present, the selection algorithm only considers the problem size to select the fastest implementation. In the future, we plan to extend the set of user template arguments and attributes to allow users to specify other kinds of restrictions, such as memory usage or energy consumption. This will allow the selector to make multi-objective optimizations.



(a) Scenario using fixed sizes.

(b) Scenario using a range of sizes.

Figure 2: Execution time of the `dgemm` operation for different problem sizes and implementations. Note the region boundaries set by the selector module.

5 Experimental evaluation

In this section, we evaluate the behavior of our adaptive offline implementation selector using the dense matrix-matrix multiplication (GEMM) and an image-processing application targeted to embedded systems (StereoBM). First, we evaluate the accuracy and convergence of the selector algorithm of the framework. Next, we study the adaptability to changes of the selection framework in heterogeneous platforms. Finally, we compare our solution using the OmpSs *versioning* runtime scheduler.

The platform used to assess the framework is equipped with an Intel Xeon processor, two AMD Radeon GPUs (connected via PCI Express 3.0) and an Intel Xeon Phi co-processor. Table 1 describes the details of these components. The OS running on this platform is a Linux Ubuntu 14.04 and the compiler is GCC v5.0 with the *concepts-lite* extension enabled.

Regarding the use cases, for the GEMM we employ the CPU implementation from the GSL library, while the clBLAS implementation is intended to run on all available devices. Figure 2, from the previous section, shows the execution times for these implementations using square matrix sizes ranging from 128 to 4,096. On the other hand, for the StereoBM use case, we use the sequential and Intel TBB framework using OpenCV routines versions for CPU and accelerators.

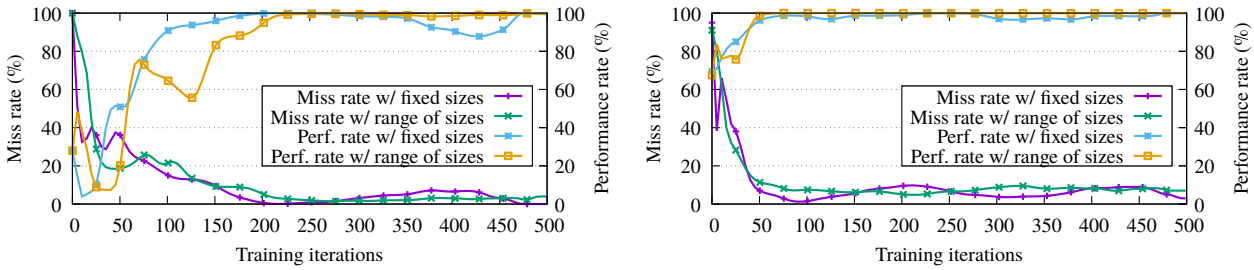
Table 1: Heterogeneous testbed platform.

	Xeon	Amd1	Amd2	Mic
Model	Intel Xeon® CPU E5-2695	AMD Radeon™ R9 290X series	AMD Radeon™ R9 285 series	Intel Xeon Phi™ 3120 series
Core clock	2.4 GHz	1030 MHz	928 MHz	1.1 GHz
Computing units	24	44 (2816)	28 (1792)	224
Memory	128 GB	4 GB	2 GB	6 GB
OpenCL version	AMD-APP OpenCL 2.0 (1642.5)			Intel OpenCL™ Runtime 14.2

5.1 Evaluation of the accuracy and performance

In this section we evaluate the accuracy and performance of the implementation selector for both use cases. Figure 3 depicts the accuracy of the decisions made by the selector and the performance attained using fixed sizes and ranges of sizes through different number of training iterations. In order to train the system, in each iteration we execute an instance of the `dgemm` kernel and the StereoBM application with random matrix sizes and image resolutions, respectively. Furthermore, to evaluate the accuracy of each training iteration we perform 100 runs using random problem sizes. Next, for each of them we compute the miss rate and the performance rate, which is obtained dividing the execution time of the fastest and the selected implementation. Note as well that calls to the `dgemm` kernel and OpenCV routines in the StereoBM application have been provided with information about the problem size or range of sizes, and processed prior the compilation phase.

As can be seen in Figure 3a, the miss rate progress for the GEMM case using both fixed and range of sizes decreases in a smooth curve until reaching, after 500 training iterations, 2 % and 5 % of the total accuracy, respectively. Similarly, the performance rate steadily increases until 100 % in both cases with fixed and range of sizes. This is mainly because the selector has already gained enough knowledge about the performance of the different implementations used. Therefore, all selections made from that point will provide good performance figures. Similarly, the miss rate progress for the StereoBM application, depicted in Figure 3b, decreases more rapidly than for the GEMM case, as it has less combinations of device-implementation available. This behavior finally leads to low miss rates: 8 % and 3 % for fixed and ranges of sizes after 500 iterations, respectively. In the same way, the performance rates at the 500-th iteration are close to 100 % for both fixed and range of sizes. In general, we observe that the miss rate for fixed problem sizes is slightly lower than for range of sizes. This is due to the size parameter provided by the user is a more accurate indicator than freely selecting a size between a given ranges. In sum, miss rates are mostly inversely proportional to performance rates, therefore, the low miss rates observed are indicators for good performance figures.



(a) Miss and performance rates for GEMM.

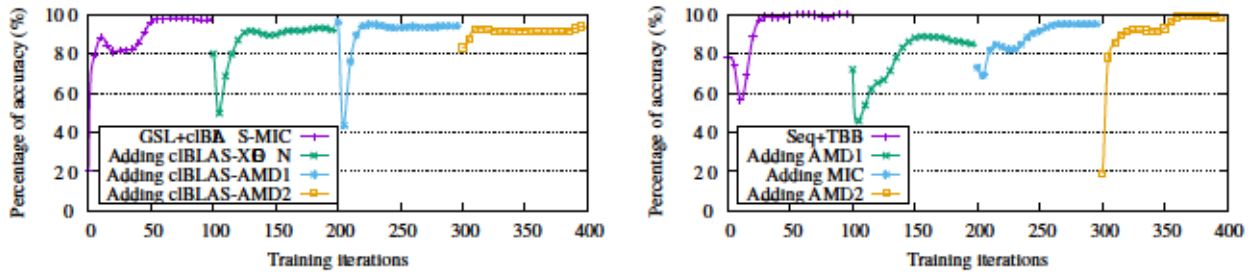
(b) Miss and performance rates for StereoBM.

Figure 3: Progress of the miss and performance rates of the selector for the GEMM and StereoBM.

5.2 Evaluation of the adaptability

Next, we evaluate the adaptability of the selector to make appropriate decisions when a new device (and implementation) is attached to the heterogeneous platform each 100 training iterations. Figure 4 analyzes the quality of the selections in this scenario for both GEMM and StereoBM use cases using range of sizes. Focusing on the GEMM case, the selector reaches about 98 % of accuracy after 100 iterations. When a new device-implementation is added, we observe that the selector needs only 20 additional iterations to converge once again and reach similar accuracy figures than before the change. Looking at the StereoBM application, we notice that when we add a new version, the knowledge considerably drops compared to the GEMM case, however the selector needs barely 25 iterations to stabilize once again. As a final remark, we observe that the accuracy reached by the selector when new devices are steadily attached to the platform is slightly higher than if all devices

are attached right at the beginning of the training process. This is given because the selector needs to train less if only one additional device is attached each time. All in all, the low numbers of additional training iterations demonstrate the ability of the framework to react in front of platform changes.



(a) Accuracy using ranges of sizes for GEMM.

(b) Accuracy using ranges of sizes for StereoBM.

Figure 4: Progress of the accuracy of the selector for the GEMM and StereoBM when adding new devices.

5.3 Comparison with alternative approaches

As stated in Section 2, several static and dynamic approaches that allow selecting different implementations of a same algorithm can be found in the literature. In this section, we validate the performance benefits of our adaptive offline implementation selector with a real runtime scheduler. Concretely, we compare our approach with the *versioning* runtime scheduler counterpart from the OmpSs programming model [3], as it offers a similar implementation selector to our static solution.

To compare our solution (AIS) with OmpSs, we developed an application composed of two 50-iteration loops that computes the matrix-matrix product (`dgemm` kernel) using square matrices of sizes 256 and 2048, respectively. Take into account that the multiplication is performed using the same `dgemm` implementations as in the previous experiments. For AIS, we annotate the `dgemm` kernel calls using the attribute `ais::size` for both matrix sizes, while in OmpSs we define different tasks for the available implementations that are annotated with the `implements` and `target` directives.

Figure 5 depicts the execution progress of this application. As observed, AIS starts from the first loop iteration selecting the implementations that perform best for the different matrix sizes. Note that AIS has been previously trained performing 100 executions of the `dgemm` kernel with random matrix sizes and the measured profiling overhead is not higher than 1%. On the contrary, OmpSs cannot be trained offline, so it makes a few trial runs of the different implementations until it finds, at runtime, the fastest one. In these cases, the single training phase of AIS pays off the OmpSs trial runs and the runtime scheduler overhead. Specifically, the OmpSs measured overhead ranges between 2% and 40% for the large and small matrix sizes, respectively. In general, AIS offers a static implementation selector that is able to train among executions and is adaptive at compile time, however OmpSs offers a more flexible alternative that requires a runtime scheduler to make decisions at the expense of non-negligible overheads.

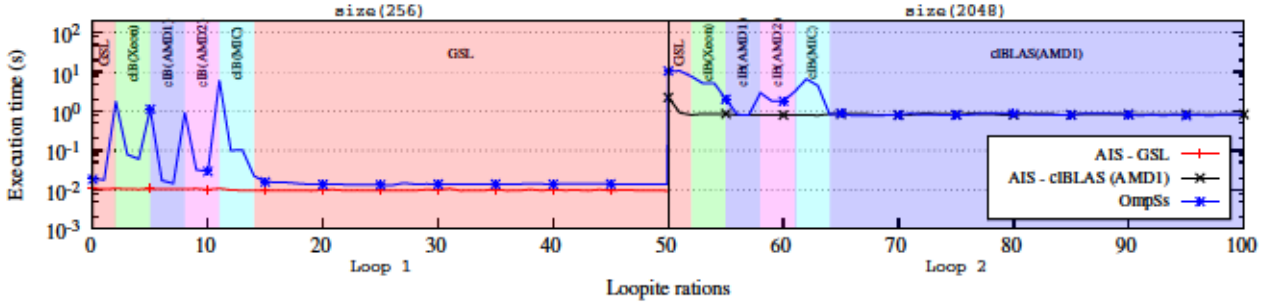


Figure 5: Execution progress of two 50-iteration loops computing the `dgemm` using AIS and OmpSs.

6 Conclusions

In this paper, we have presented an adaptive offline implementation selector for heterogeneous platform that selects automatically the tuple device-implementation delivering the best performance according to the problem size and historical information. Thanks to this approach, our framework shifts the decision-making process

at compile time, so that overheads related to dynamic scheduling approaches are also shifted in a negligible profiling process. Furthermore, our framework is hardware independent, therefore, it is possible to freely add or remove devices of the platform without incurring significant overheads. To enable portability, our implementation selector leverages two novel C++ features (attributes and concepts), inherent to the standard C++ programming language.

To evaluate the benefits of this framework, we analyzed the performance and accuracy of the tool using two different use cases: the general matrix-matrix multiplication and an image-processing application. The experimental results prove that the selector enhances performance while minimizes efforts to tune applications targeted to heterogeneous platforms. Furthermore, the results also show that our framework delivers comparable performance figures with respect to the OmpSs *versioning* runtime scheduler.

As future work, we plan to extend the set of C++ attributes in order to allow users specify other kinds of restrictions, such as memory usage or energy consumption. Furthermore, we also aim to incorporate a static partitioning module for supporting multiple devices in shared and distributed memory systems. Another goal is to integrate the attribute-based selector tool as part of the Clang C++ compiler.

[2]

References

[1] M.I. Daoud and N. Kharma. Efficient compile-time task scheduling for heterogeneous distributed computing systems. In *Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on*, volume 1, pages 9 pp.–, 2006.

[2] Usman Dastgeer, Lu Li, and Christoph Kessler. *Advanced Parallel Processing Technologies: 10th International Symposium, APPT 2013, Stockholm, Sweden, August 27-28, 2013, Revised Selected Papers*, chapter Adaptive Implementation Selection in the SkePU Skeleton Programming Library, pages 170–183. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[3] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21:173–193, 2011-03-01 2011.

[4] Brian Gough. *GNU Scientific Library Reference Manual - Third Edition*. Network Theory Ltd., 3rd edition, 2009.

[5] V. Kindratenko and P. Trancoso. Trends in High-Performance Computing. *Computing in Science Engineering*, 13(3):92–95, May 2011.

[6] Jens Maurer and Michael Wong. Towards support for attributes in C++ (Revision 6). In *JTC1/SC22/WG21 - The C++ Standards Committee*, 2008. N2761=08-0271.

[7] nVidia. *cuBLAS Library User Guide*. nVidia, v5.0 edition, October 2012.

[8] REPARA project. Target Platform Description Specification. `https://repara-project.eu/wp-content/uploads/2014/04/ICT-609666-D3.1.pdf`, February 2014.

[9] Luis Miguel Sanchez, David del Rio Astorga, Manuel F. Dolz, and Javier Fernández. CID: A Compile-time Implementation Decider for Heterogeneous Platforms based on C++ Attributes. In *2016 Intl IEEE Scalable Computing and Communications*, pages 1149–1156, 2016.

[10] LuisMiguel Sanchez, Javier Fernandez, Rafael Sotomayor, Soledad Escolar, and J.Daniel. Garcia. A Comparative Study and Evaluation of Parallel Programming Models for Shared-Memory Parallel Architectures. *New Generation Computing*, 31(3):139–161, 2013.

[11] Jie Shen, A.L. Varbanescu, and H. Sips. Look before you leap: Using the right hardware resources to accelerate applications. In *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS), 2014 IEEE Intl Conf on*, pages 383–391, Aug 2014.

[2]This work has been partially supported by the Spanish "Ministerio de Economía y Competitividad" under the project grant TIN2016-79637-P "Towards Unification of HPC and Big Data Paradigms" and the EU Projects ICT 644235 "REPHRASE: REfactoring Parallel Heterogeneous Resource-Aware Applications" and the FP7 609666 "REPARA: Reengineering and Enabling Performance And poweR of Applications'.

[12] Andrew Sutton. Working Draft, C++ Extensions for Concepts. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4040.pdf`, May 2014.

[13] Andrew Sutton and Bjarne Stroustrup. Concepts Lite: Constraining Templates with Predicates. `http://concepts.axiomatics.org/~ans/concepts-lite.pdf`, January 2012.

[14] Wen Jun Tan, Wai Teng Tang, R.S.M. Goh, S.J. Turner, and Weng-Fai Wong. A code generation framework for targeting optimized library calls for multiple platforms. *Parallel and Distributed Systems, IEEE Transactions on*, 26(7):1789–1799, July 2015.

[15] Zheng Wang, Dominik Grewe, and Michael F. P. O'boyle. Automatic and Portable Mapping of Data Parallel Programs to OpenCL for GPU-Based Heterogeneous Systems. *ACM Trans. Archit. Code Optim.*, 11(4):42:1–42:26, December 2014.